CS162  
Operating Systems and  
Systems Programming  
Lecture 11  

Page Allocation and Replacement  

March 3, 2014  
Anthony D. Joseph  
http://inst.eecs.berkeley.edu/~cs162

Post Project 1 Class Format  
• Mini quizzes after each topic  
  – Not graded!  
  – Simple True/False  
  – Immediate feedback for you (and me)  

• Separate from pop quizzes

Quiz 11.1: Caches & TLBs  
• Q1: True _ False _ Associative caches have fewer compulsory misses than direct mapped caches  
• Q2: True _ False _ Two-way set associative caches can cache two addresses with same cache index  
• Q3: True _ False _ With write-through caches, a read miss can result in a write  
• Q4: True _ False _ A TLB caches translations to virtual addresses

Quiz 11.1: Caches & TLBs  
• Q1: True _ False X Associative caches have fewer compulsory misses than direct mapped caches  
• Q2: True X False _ Two-way set associative caches can cache two addresses with same cache index  
• Q3: True _ False X With write-through caches, a read miss can result in a write  
• Q4: True _ False X A TLB caches translations to virtual addresses
Review: Two-level page table

Virtual Address: 10 bits P1 index 10 bits P2 index 12 bits Offset

Physical Address: Offset

PageTablePtr

4 bytes

IA64: 64-bit addresses: Six-level page table?!?

64bit Virtual Address: 7 bits P1 index 9 bits P2 index 9 bits P3 index 9 bits P4 index 9 bits P5 index 12 bits P6 index Offset

No!

Too slow
Too many almost-empty tables

IA64: Inverse Page Table (IPT)

Idea: index the page table by physical pages instead of VM

X86_64: Four-level page table!

48-bit Virtual Address: 9 bits P1 index 9 bits P2 index 9 bits P3 index 9 bits P4 index 12 bits Offset

4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

VMpage0, proc0
VMpage1
VMpage2
VMpage3

Process id 0
Virtual memory

Physical memory

in 4kB pages
Page numbers in red

Physical Address: (40-50 bits)
• Need an associative map from VM page to IPT address:
  Use a hash map

IPT address translation

Pros:
• Page table size naturally linked to physical memory size
• Only two memory accesses (most of the time).
• Shouldn’t need to page out the page table
• Hash function can be very fast if implemented in hardware

Cons:
• Can’t (easily) share pages
• Have to manage collisions, e.g., by chaining, which adds memory accesses

IA64: Inverse Page Table (IPT)

Quiz 11.2: Address Translation

Q1: True  False  Paging does not suffer from external fragmentation
Q2: True  False  The segment offset can be larger than the segment size
Q3: True  False  For paging, to compute the physical address, you add physical page # and offset
Q4: True  False  Uniprogramming doesn’t provide address protection
Q5: True  False  A virtual address space is always larger than physical address space
Q6: True  False  Inverted page tables keep fewer entries than multi-level page tables
Quiz 11.2: Address Translation

- Q1: True _ False _ Paging does not suffer from external fragmentation
- Q2: True _ False _ The segment offset can be larger than the segment size
- Q3: True _ False _ For paging, to compute the physical address, you add physical page # and offset
- Q4: True _ False _ Uniprogramming doesn’t provide address protection
- Q5: True _ False _ A virtual address space is always larger than physical address space
- Q6: True _ False _ Inverted page tables keep fewer entries than multi-level page tables

Review: Translation Lookaside Buffer

Review: Cache

Goals for Today

- Page Replacement Policies
  - FIFO
  - LRU
  - Clock Algorithm

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.
Modern programs require a lot of physical memory
- DRAM per system growing faster than 25%-30%/year
But they don’t use all their memory all of the time
- 90-10 rule: programs spend 90% of their time in 10% of their code
- Wasteful to require all of user’s code to be in memory
Solution: use main memory as cache for disk/SSD

Demand Paging is Caching
Since Demand Paging is Caching, we must ask:
- What is the block size?
- What is the organization of this cache (i.e., direct-mapped, set-associative, fully-associative)?
- How do we find a page in the cache?
- What is page replacement policy? (i.e., LRU, Random, …)
- What happens on a miss?
- What happens on a write? (i.e., write-through, write-back)

Demand Paging Mechanisms
PTE helps us implement demand paging
- Valid ⇒ Page in memory, PTE points at physical page
- Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
Suppose user references page with invalid PTE?
- Memory Management Unit (MMU) traps to OS
  » Resulting trap is a "Page Fault"
- What does OS do on a Page Fault?:
  » Choose an old page to replace
  » If old page modified ("D=1"), write contents back to disk
  » Change its PTE and any cached TLB to be invalid
  » Load new page into memory from disk
  » Update page table entry, invalidate TLB for new entry
  » Continue thread from original faulting location
  » TLB for new page will be loaded when thread continued!
  » While pulling pages off disk for one process, OS runs another process from ready queue
  » Suspended process sits on wait queue

Steps in Handling a Page Fault
Demand Paging Example

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - $EAT = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time}$
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose $p = \text{Probability of miss}, 1-p = \text{Probable of hit}$
  - Then, we can compute $EAT$ as follows:
    $$EAT = (1 - p) \times 200\text{ns} + p \times 8,000,000\text{ns}$$
    $$= 200\text{ns} + p \times 7,999,800\text{ns}$$
- If one access out of 1,000 causes a page fault, then $EAT = 8.2 \mu s$:
  - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
  - $EAT < 200\text{ns} \times 1.1 = p < 2.5 \times 10^{-6}$
  - This is about 1 page fault in 400,000!

What Factors Lead to Misses?

- **Compulsory Misses:**
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - Prefetching: loading them into memory before needed
    - Need to predict future somehow! More later.
- **Capacity Misses:**
  - Not enough memory. Must somehow increase size.
  - Can we do this?
    - One option: Increase amount of DRAM (not quick fix!)
    - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- **Conflict Misses:**
  - Technically, conflict misses don’t exist in virtual memory, since it is a “fully-associative” cache
- **Policy Misses:**
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Better replacement policy

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk
    - Must keep important pages in memory, not toss them out
- **FIFO (First In, First Out)**
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- **MIN (Minimum):**
  - Replace page that won’t be used for the longest time
  - Great, but can’t really know future...
  - Makes good comparison case, however
- **RANDOM:**
  - Pick random page for every replacement
  - Typical solution for TLB’s. Simple hardware
  - Unpredictable

Replacement Policies (Con’t)

- **FIFO:**
  - Replace page that has been in for the longest time.
  - Be “fair” to pages and give them equal time
  - Bad idea because page use is not even. We want to give more time to heavily used pages
- **How to implement FIFO? It’s a queue (can use a linked list)**

- Oldest page is at head
- When a page is brought in, add it to tail.
- Eject head if list longer than capacity
Replacement Policies (Con’t)

- LRU (Least Recently Used):
  - Replace page that hasn’t been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future
  - Seems like LRU should be a good approximation to MIN

- How to implement LRU? Use a list?
  - LRU page is at head
  - When a page is used for the first time, add it to tail
  - Eject head if list longer than capacity

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B

- Consider FIFO Page replacement:
  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

FRAMES

<table>
<thead>
<tr>
<th>Ref</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

- Updates are happening on page use, not just swapping
- List structure requires extra pointers c.f. FIFO, more updates
- In practice, people approximate LRU (more later)
Example: MIN

- Suppose we have the same reference stream:
  - A B C A B D A B C B
- Consider MIN Page replacement:

<table>
<thead>
<tr>
<th>Ref:</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

- MIN: 5 faults
- Look for page not referenced farthest in future.
- What will LRU do?
- Same decisions as MIN here, but won’t always be true!

When will LRU perform badly?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

<table>
<thead>
<tr>
<th>Ref:</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

- Every reference is a page fault!
- MIN Does much better:

<table>
<thead>
<tr>
<th>Ref:</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

Adding Memory Doesn’t Always Help Fault Rate

- One desirable property: When you add memory the miss rate goes down
  - Does this always happen?
  - Seems like it should, right?
- No: Belady’s anomaly
  - Certain replacement algorithms (FIFO) don’t have this obvious property!

Graph of Page Faults Versus The Number of Frames

- Does adding memory reduce number of page faults?
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Called Belady’s anomaly)

<table>
<thead>
<tr>
<th>Page:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>B</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- After adding memory:
  - With FIFO, contents can be completely different
  - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page
Administrivia

• Should be working on Project #2:
  – Design doc due Thu 3/6 at 11:59PM

• Midterm #1 is Wed March 12 4-5:30pm in 245 Li Ka Shing (A-L) and 105 Stanley (M-Z)
  – Closed book, double-sided handwritten page of notes, no calculators, smartphones, Google glass etc.
  – Covers lectures #1-12, readings, handouts, and projects 1 and 2
  – Review session: 245 Li Ka Shing Mon March 10 5:30-7:30pm

• Class feedback is always welcome!
  – https://www.surveymonkey.com/s/ZFDLLYL

Implementing LRU & Second Chance

• Perfect:
  – Timestamp page on each reference
  – Keep list of pages ordered by time of reference
  – Too expensive to implement in reality for many reasons

• Second Chance Algorithm:
  – Approximate LRU
  – Replace an old page, not the oldest page
  – FIFO with "use" bit

• Details
  – A "use" bit per physical page
    » set when page accessed
  – On page fault check page at head of queue
    » If use bit=1 → clear bit, and move page to tail (give the page second chance!)
    » If use bit=0 → replace page
  – Moving pages to tail still complex

Second Chance Illustration

• Max page table size 4
  – Page B arrives
  – Page A arrives
  – Access page A
  – Page D arrives
  – Page C arrives

  first loaded page ↓  B u:0 A u:1  D u:0  C u:0  last loaded page ↓
Second Chance Illustration

• Max page table size 4
  – Page B arrives
  – Page A arrives
  – Access page A
  – Page D arrives
  – Page C arrives
  – Page F arrives

First loaded page ↓  last loaded page ↓

  B u:0    A u:1    D u:0    C u:0


Second Chance Illustration

• Max page table size 4
  – Page B arrives
  – Page A arrives
  – Access page A
  – Page D arrives
  – Page C arrives
  – Page F arrives

First loaded page ↓  last loaded page ↓

  A u:1    D u:0    C u:0    F u:0


Second Chance Illustration

• Max page table size 4
  – Page B arrives
  – Page A arrives
  – Access page A
  – Page D arrives
  – Page C arrives
  – Page F arrives

First loaded page ↓  last loaded page ↓

  A u:1    D u:1    C u:0    F u:0


Second Chance Illustration

• Max page table size 4
  – Page B arrives
  – Page A arrives
  – Access page A
  – Page D arrives
  – Page C arrives
  – Page F arrives
  – Access page D
  – Page E arrives

First loaded page ↓  last loaded page ↓

  A u:1    D u:1    C u:0    F u:0

Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives

Clock Algorithm

- Clock Algorithm: more efficient implementation of second chance algorithm
  - Arrange physical pages in circle with single clock hand
- Details:
  - On page fault:
    - Check use bit: 1 → used recently; clear and leave it alone
    - 0 → selected candidate for replacement
  - Advance clock hand (not real time)
  - Will always find a page or loop forever?
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page

- Page B arrives
- Page A arrives
- Access page A

Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page

- Page B arrives
- Page A arrives
- Access page A
- Page D arrives

Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page

- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives
- Page F arrives
- Access page D
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives

Clock Algorithm: Discussion

- What if hand moving slowly?
  - Good sign or bad sign?
    » Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set

N\textsuperscript{th} Chance version of Clock Algorithm

- N\textsuperscript{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    » 1⇒clear use and also clear counter (used in last sweep)
    » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approx to LRU
    » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    » Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    » Clean pages, use N=1
    » Dirty pages, use N=2 (and write back to disk when N=1)

Clock Algorithms: Details

- Which bits of a PTE entry are useful to us?
  - Use: Set when page is referenced; cleared by clock algorithm
  - Modified: set when page is modified, cleared when page written to disk
  - Valid: ok for program to reference this page
  - Read-only: ok for program to read page, but not modify
    » For example for catching modifications to code pages!
- Do we really need hardware-supported “modified” bit?
  - No. Can emulate it (BSD Unix) using read-only bit
    » Initially, mark all pages as read-only, even data pages
    » On write, trap to OS. OS sets software “modified” bit, and marks page as read-write.
    » Whenever page comes back in from disk, mark read-only
Clock Algorithms Details (cont’d)

- Do we really need a hardware-supported “use” bit?
  - No. Can emulate it using “invalid” bit:
    » Mark all pages as invalid, even if in memory
    » On read to invalid page, trap to OS
    » OS sets use bit, and marks page read-only
  - When clock hand passes by, reset use bit and mark page as invalid again

Quiz 11.3: Demand Paging

- Q1: True _ False _ Demand paging incurs conflict misses
- Q2: True _ False _ LRU can never achieve a higher hit rate than MIN
- Q3: True _ False _ The LRU miss rate may increase as the cache size increases
- Q4: True _ False _ The Clock algorithm is a simpler implementation of the Second Chance algorithm
- Q5: Assume a cache with 100 pages. The number of pages that the Second Chance algorithm may need to check before finding a page to evict is at most ___

Quiz 11.3: Demand Paging

- Q1: True _ False _ Demand paging incurs conflict misses
- Q2: True _ False _ LRU can never achieve a higher hit rate than MIN
- Q3: True _ False _ The LRU miss rate may increase as the cache size increases
- Q4: True _ False _ The Clock algorithm is a simpler implementation of the Second Chance algorithm
- Q5: Assume a cache with 100 pages. The number of pages that the Second Chance algorithm may need to check before finding a page to evict is at most 101

Summary (1/2)

- Demand Paging:
  - Treat memory as cache on disk
  - Cache miss ⇒ find free page, get page from disk
- Transparent Level of Indirection
  - User program is unaware of activities of OS behind scenes
  - Data can be moved without affecting application correctness
- Replacement policies
  - FIFO: Place pages on queue, replace page at head
    » Fair but can eject in-use pages, suffers from Belady’s anomaly
  - MIN: Replace page that will be used farthest in future
    » Benchmark for comparisons, can’t implement in practice
  - LRU: Replace page used farthest in past
    » For efficiency, use approximation
Summary (2/2)

• Clock Algorithm: Approximation to LRU
  – Arrange all pages in circular list
  – Sweep through them, marking as not “in use”
  – If page not “in use” for one pass, than can replace